Thesis Project I

Making scope explorable in Software Development Environments to support understanding and reduce errors

Author:
Tim von Oldenburg

Supervisor:
Lars Holmberg

May 22, 2014

Abstract

Research Question: *How can an IDE support the understanding and exploration of scope & context in a programming language to deepen understanding and help prevent errors?*

- IDE giving semantic support in code creation
- Understanding asynchronous control flow in IDEs // nope!!!
- Understanding and exploring scope and context
- Goals: understanding, less errors/mistakes

Acronyms

API Application Programming Interface

APM Atom Package Manager

AST Abstract Syntax Tree

CSS Cascading Stylesheets

DOM Document Object Model

HTML Hypertext Markup Language

IDE Integrated Development Environment

JIT Just-In-Time

JVM Java Virtual Machine

OSS Open Source Software

PARC Palo Alto Research Centre

UCD User-Centered Design

UI User Interface

WYSIWYG What You See Is What You Get

Contents

1 Introduction

Creating computer programs is a difficult and complex task. Nowadays, software developers can rely on a number of tools to help them do their tasks. Often, these tools are integrated by a so-called Integrated Development Environment (IDE).

Ansatz: identify characteristics of well-integrated development tools, apply to the problem of scope, implement, validate. Passt das?

So-called static analysis tools are especially helpful in maintaining code quality. - devs rely on tools - ides integrate tools - static analysis tools can help at author-time, like linters - problem in many languages: scope - can ides help? - alternative way of looking at a program - Target group: professional (js) developers

Scope is a phenomenon of computer programs related to the validity of variables. By looking at the structure of scope, a program can be explored from a different perspective, and certain pitfalls that lead to programming errors can be uncovered. Scope structure can be explored using static analysis tools, and therefore be usable during author-time already.

What will be my knowledge contribution?

-anticipated:knowledge contribution: how do i evaluate designs like this, e.g. ide tools, dev tools, for a user groups like this?

-not anticipated: not anticipated knowledge contribution: using the ixd approach for an open source project yielded new results, a more open field, than os approaches usually do (they copy more and adapt and stuff)

- Which characteristics are good for integrating language tools into the dev workflow?
- Best way to integrate code evaluation, a new way of looking at software, into the creation process

Following a User-Centered Design (UCD) process, the project described in this thesis targets the needs of professional developers with advanced experience. The final design is built for the JavaScript programming language, but the concept of scope presents difficulties in nearly every language in use. The knowledge gained during the process is thus expected to be applicable to other programming languages as well.

Integrating a solidly implemented, high-level prototype in the Atom text editor will demonstrate the value of the concept. It will be validated by means of both quantitative and qualitative data using analytics, general feedback on the web, and interviews.

The goal of this thesis is twofold. On the one hand, the author wants to identify characteristics of well-integrated software development tools.

Mein roter Pfaden, eins von - Loose Integration von language tools in text editoren, statt full-fledged IDEs - Static analysis tools inline, statt auf der command line - IDE vs code/text editor

2 Theoretical Grounding

This chapter will introduce the research done prior to the design. It will present a short history of IDEs and list typical User Interface (UI) design patterns in IDEs. Finally, the principle of *scope* in programming languages will be explained.

2.1 History and Purpose of Integrated Development Environments

> „A programming environment is a user interface for understanding a program.“ — Bret Victor (2012)

Software development environments have been predecessed by general text editors, starting with several projects at the Xerox Palo Alto Research Centre (PARC). Douglas Engelbart created the text editor for the NLS system (oNLine System) which allowed What You See Is What You Get (WYSIWYG) style editing. In the *Gypsy* text editor, Larry Tesler first integrated modeless moving of text, which is known as *Copy & Paste* (Moggridge 2007). Text editors with those functionalities are now the core of any software development environment.

Later, while working with Alan Kay, Tesler created the first class browser for the Smalltalk programming language. Class browsers are used to look at programs not as of textual source code, but as of logical entities of a programming language (for example classes and methods). The Smalltalk class browser was therefore the first software specifically written for creating software, and a predecessor to any modern development environment.

Integrated Development Environments integrate text editors (due to their specific purpose also referred to here as *code editors*) with other software development tools. Typically, those tools include compilers, build systems, syntax highlighters, autocompletion, debuggers, and symbol browsers. The first IDE is said to be *Maestro I* by Softlab, a whole terminal dedicated to integrating various development tasks (*Interaktives Programmieren als System-Schlager* 1975).

2.1.1 IDEs compared to Text Editors

It is difficult to delimit the term „Integrated Development Environment“ and contrast it with text editor that are mainly used for programming. Baxter-Reynolds formulates a basic definition:

> „What the different is between a text editor and an IDE – to me at least – is that an IDE understands the language, whereas the text editor understands text.“ (2011)

In his article, Baxter-Reynolds tries to make a point against the use of text editors for programming by stating that an IDE brings „forward an understanding of the underlying language and the structure of code, and puts it front-and-centre in your working environment.“ (2011) While certainly being correct with this point, he ignores situations where the „understanding of the underlying language and the structure of code“ is either not wanted[1] or not possible to achieve.

The latter is often the case in web front-end development, according to Lynch (2011). Through working with lots of different file types and programming languages, neither of which dictates a certain structure (in opposition to many static languages like Java), the understanding an IDE can have about the structure of the code is limited. Lynch also states that IDEs „tend to be built with a workflow in mind“, therefore being seen as opinionated.

In other words, IDEs and text editors seem to follow different, contradirectional approaches. While the latter is built around a central paradigm (text editing) and usually comes with a minimal program core that is extendable to personal likes, IDEs tend to offer everything „out of the box“ as a one-stop solution.

For this thesis, the distinction only plays a subordinate role, as most of the concepts and ideas discussed here can be applied to both kinds of software. However, it is important to clarify that both are adressed when using, interchangably, any of the following terms: *Integrated Development Environment (IDE)*, *development environment*, *software development environment*, *programming environment*.

[1] For example, because it may collide with other features that have a higher priority for the respective developer.

2.2 UI and Interaction Patterns in IDEs

Many User Interface patterns found in IDEs are general, well-known UI patterns adapted to a specific purpose. This section gives an overview on interaction patterns in IDEs that are relevant to this thesis.

2.2.1 User Interface Patterns

Code Editor Central to every IDE, a code editor is a specialized text editor, used for reading and writing program code. It typically features a *gutter* (see below) and Syntax Highlighting. In opposition to the text editor of a word processor, code editors are not rich text editors. They also display a monospaced font, which allows to see the editor content as a grid of rows and columns. With evenly-spaced columns, due to the monospaced font, code formatting and line indentation[1] is made consistent.

Gutter The gutter is part of the code editor and describes the narrow space next to the actual code (usually to the left). Gutters are mainly used to display line numbers (important for navigation and debugging), but some provide more advanced features, for example setting breakpoints[2], indicating errors in the code through symbols, showing version control information, or allowing to fold code away in order to either focus or get an overview.

Panel (sidebar) A panel is rectangular UI area used to group together interface element of similar functionality or other commonalities together. Often, panels are used on the edges of application windows; if they are on the left or right side, they may be called *sidebar*. Panels that host a great number of program functionalities are often called *toolbar*. Some applications implement *dockable* panels, which can be moved around and snapped to different areas on the screen. Another common characteristic is that panels can be resized and *toggled*, i.e. shown and hidden, on demand.

Status bar The status bar is known from many programs, for example web browsers and word processors. It is a small bar (about one text line of height) at the bottom of the program window, usually spanning the whole window width. It is mainly used to display status information and quickly switch between different application modes (for example „insert“ and „overwrite“ in word processors).

[1] In many programming languages, line indentation is an important concept, either as a core syntactical concept or for the sake of readability.

[2] A feature of the debugger; when set, the program stops at the specified line to allow step-by-step investigation.

2.2.2 Interactional patterns

Navigation Usually, code can be both browsed and searched for from different perspectives.

For browsing, most IDEs have a built-in file browser. IDEs that have the respective understanding of code structure can also offer a more *logical* way of navigating, for examply by symbolic entities like modules, classes and methods. Those are usually listed in a symbol browser or class browser. In the Eclipse IDE, the file browser and symbol browser are combined into one component, called the *project explorer*.

IDE facilities for searching work analoguously. Files within a project can be searched for by their name or their content. If the IDE knows about the symbols of a programming language, those can usually be searched for as well. Additionally, some IDEs like Eclipse allow the user to right click on a method call and jump to its definition source file, if available.

Modes In most IDEs, UI elements can be shown or hidden, sometimes even positioned anywhere on the screen. The Eclipse IDE even allows the creation of completely different UI configurations, so-called *perspectives*. Usually, perspectives are build for a certain task, e.g. developing or debugging. Text editors like Sublime Text and Atom[1] support a so-called *distraction-free mode*, in which all User Interface elements are hidden except the editor itself.

Input todo

Execution and Debugging todo

2.3 Relevant Programming Concepts

The following section presents concepts of programming and programming languages that are important to the topic of this thesis. Whereas most of the concepts apply to a wide range of programming languages, *JavaScript* was chosen as an exemplary language both to explain the concepts as well as the target language of prototyping as described in chapter 5: *Design*. The reasons for this choice are my familiarity with the language, as well as the fact that JavaScript is one of the most ubiquituous languages used due to its role in the world wide web and its implementation in web browsers, respectively.

[1] In Atom, this has to be installed through a package: `https://atom.io/packages/zen`

2.3.1 Program Lifecycle

The lifecycle of a computer program consists of different phases, the most relevant of which are described briefly in this section.

Author-time shall be the phase during which a program is written, read, understood, and edited. There is no canonical definition or common name for this class of activities around source code, which is we define *author time* as the time separate from run time in which a program author (e.g. a developer) deals directly with its code. An alternative name for author-time may be *edit-time*, *creation-time* (Simpson 2014) or *construction* (McConnell, Steve 2004).

Compile-time is the phase in which program code is translated (compiled) into native machine code or an intermediate representation (e.g. Java Bytecode in the case of the Java Virtual Machine (JVM)). This process generally consists of lexical analysis, parsing and code generation.

Run-time is the phase during which a program is executed. In some interpreted languages, Just-In-Time (JIT) compilation[1] leads to a convergence of compile-time and run-time, which makes the distinction harder. *Run time errors* are errors happening during run-time that could not be detected during compilation (for example if they depend on user input).

Debugging is the process of identifying and eliminating software errors, so-called *bugs*. This activity is usually supported by a specialized software called a *debugger*. The debugger allows to hook into a program during run-time through so-called *breakpoints* and step through each statement individually. At all times, the debugger can expose the values of variables in the respective context.

This thesis and the according prototype mainly address the author-time phase, during which so-called static analysis can be performed.

2.3.2 Scope & Context

In computer programming, data is usually addressed through variables. At some point in the program, a variable is *declared*, i.e. its existence is made known to the program. However, in most programming languages, a variable declaration in some part of the program does not necessarily

[1] Just-in-Time compilation is the compilation of code immediately before its execution, instead of during a preliminary compilation phase.

make the variable accessible from *all other* parts of the program. The area in which the variable is accessible is called its *scope*.

According to Simpson (2014), *scope* is „the set of rules that determines where and how a variable (identifier) can be looked-up“ and therefore be accessed and used. The specifics of „where and how“ depend on the respective programming language. Most modern languages implement *lexical scope*, which means that the scope of a variable depends on the position of its declaration in the actual source code. In other words, where in the source text a variable is declared defines also where it is usable and accessible.[1] Lexical scope also means that scope is defined during author-time already, and can thus be analyzed early on. In contrast, the *this* keyword in JavaScript is a run-time phenomenon; its value cannot be known during author-time.

- scope vs context

Nested scope & variable lookup

Scope is a hierarchical concept: in many programming languages, scope can be nested by creating a scope *within* another scope. Consequently, we will use the following definitions throughout this document:

Child scope A scope `b` created immediately within another scope `a` is a child scope to `a`.

Descendant scope Any scope nested inside of a scope `a` is descendant to scope `a`.

Parent scope The scope in which an immediate child scope is created is its parent scope.

Ancestor scope If scope `b` is a descendant to scope `a`, `a` is an ancestor of scope `b`.

In JavaScript, scope nesting is an important concept for variable lookup. When the JavaScript engine encounters an identifier, it looks for this identifier in the current chain of scopes. For example, if a variable is used in a scope `a`, the JavaScript engine first looks for its declaration in the immediate scope, `a`. However, if it cannot be found in the immediate scope, the next outer scope (the parent scope of `a`) is consulted, continuing the hierarchy of ancestors up until the outermost (global) scope has been reached. In other words: A variable is valid in the scope it was created, as well as in all nested (descendant) scopes. This circumstance leads to the phenomenon of shadowing, which is described later in this chapter. As this way of looking up variables is executed *each time a variable is encountered*, it can have impacts on the performance as

[1] The complementing concept, *dynamic scope*, is not relevant to this thesis.

well, especially if the encountered variable is defined in a scope many levels higher in the scope chain.

Nested scope can best be illustrated by the following figure:

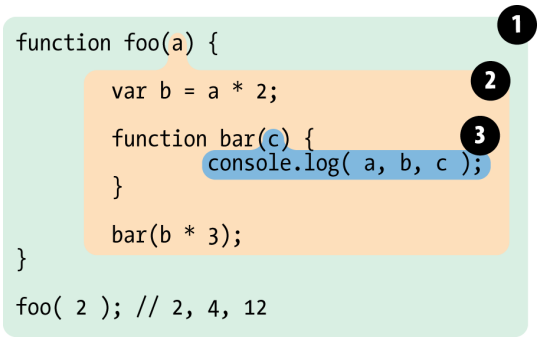

Figure 2.1: Nested scope (Simpson 2014)

The function `foo` is defined *in* the global scope (1) (see next section), and is therefore accessible from all parts of this program. `foo` itself defines a new scope (2) which includes the identifiers `a`, `b` and `bar`. `bar` defines a new scope (3) within `foo`, defining only the identifier `c`. As can be seen, the innermost scope (3) has access to its own identifiers, as well as to the ones defined in its containing scope (2).

Scoping Models

As mentioned above, the rules for when a new scope is defined differ depending on the programming language. Usually, a language implements multiple of the following rules.

Global scope Variables that are accessible from *any point* in the program are in the global scope. The original BASIC programming language only implemented global scope.

Block scope Any logical block, often denoted by containing curly braces (`{` and `}`), will create a new scope. This is the case in the C programming language, amongst others.

Function scope Any function definition defines a new scope. Parameters of the function are part of this newly defined scope, as well as variables and function defined within that function. JavaScript implements function scope.

Expression scope A variable's scope is limited to a single expression. This useful for very short-living, temporary variables. It is implemented by many functional languages, for example Python and ECMAScript 6.

JavaScript, as of ECMAScript 5, implements only global and function scope[1].

In JavaScript, the run-time environment defines what is in the global scope. The JavaScript engines in web browsers usually provide access to the Document Object Model (DOM) through the `document` object, whereas Node.js provides the `require` function to include CommonJS-style modules.

In contrast, the Java programming language implements block scope, but no global scope. - Functions define a new scope; blocks do not (in JavaScript)

Common scoping problems

The following are common phenomena that arise through scoping and may be the cause of problems and misconceptions. Though being typical for JavaScript, many of those problems can arise in other programming languages, in the same or similar form, as well.

These phenomena can in most cases be either helpful or hindering, and thus be desired or undesired. The goal of the concept developed in this thesis is to make the developer recognize those phenomena during author-time, and thus avoid misconceptions and reduce errors.

Hoisting is the implicit process, as done by the JavaScript engine, of moving variable and function declarations „from where they appear in the flow of the code to the top of the code“ (Simpson 2014). By code, Simpson refers to the scope block. Any variable declaration inside a scope block is hoisted to the top of the scope block.

```
function foo() {
  a = 2;
  var a;
  console.log( a );
}
```

The above code is actually processed as:

```
function foo() {
  var a;
  a = 2;
  console.log( a );
}
```

[1] There are exceptions through the keywords ‘with‘ and ‘except‘ and the function ‘eval‘.

The variable declaration of `a` is moved, or „hoisted“, to the top of the scope block of `foo`. Hoisting can impose unexpected behaviour, especially when declaring variables of the same name in nested scopes.

Closure is a common phenomenon in JavaScript programs, and is widely used, though being generally seen as hard to understand. Citing Simpson, closure is „when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.“ (2014) As functions are first-class objects in JavaScript, they can be passed around like variables, for example as callbacks. A function can also return another function. However, JavaScript works with *lexical scope* and, according to the nesting rules presented before, a function must always have access to its ancestor scopes. Thus, when a function is being returned or passed as a callback, an instance of the whole scope chain is returned or passed along with the function. In other words, the function „closes“ or „forms a closure“ over its ancestor scopes. In most cases, this behaviour is desired. Anyway it is important to recognize closures, as they may impact performance: the closed-over scopes have to stay in memory as long as a reference to the closure exists. Closure may also lead to unexpected behaviour, for example if a variable defined outside of a closure is used inside of it (see Simpson (2014, Ch. 5) for examples).

Shadowing is a consequence of nested scopes. If a variable (1) is defined in an ancestor scope, and a new variable (2) of the same name is defined in a descendant scope, the descendant scope has no access to (1). This is due to the mechanism of variable lookup explained above. Variable (1) is said to be *shadowed* by variable (2). As with most of the phenomenons listed here, this can either be desired or unwanted behaviour. A good solution to avoid shadowing is choosing different variable names throughout nested scopes.

Implicit variable declaration JavaScript allows for the creation of variables and object properties in an implicit way (*silently*). Instead of declaring a variable using a `var` statement, they can as well just be used without prior declaration, for example like this:

```
i = 3;
```

Variables used without prior declaration are implicitly declared in the *global scope*[1]. As this is usually unwanted behaviour, it is considered good practice to always declare variables explicitly. However, this problem is already addressed by linters (see 4.2: *Solutions to analogous problems*).

[1] ECMAScript 5's *strict mode* considers this an error.

Lookup performance The variable lookup through scope chains, as described above, can have an impact on the performance of an application. Each time a variable is encountered, the JavaScript engine performs the lookup process, navigating from the bottom of the scope chain upwards until it is found. If a variable, which is defined in an ancestor scope (the global scope, for example), is accessed within a deeply nested scope, the lookup process slows down the execution of the program, as shown by Castorina (2014). He furthermore suggests to cache the variable in a „closer“ scope, if possible.

3 Methodology & Process

3.1 Process

- Ideation & Concept
- Prototyping: esprima, atom, brackets, devtools?
- User Testing, Probe (1 week)
- Interview, evaluation in quantity and quality

4 Exploration

This chapter presents the exploration phase of the design process. Through an initial survey and a sequence of interviews, a whole range of problems was explored before a single problem, scope, was put into focus. By looking at solutions to conceptually similar problems and an ideation phase, a series of concepts was sketched out.

4.1 Survey & Interviews

To form a general understanding of how IDEs and some of their specific features are used, an online survey targeted towards professional developers was created. The survey ran in April 2014 over the course of two weeks and yielded answers from 45 participants.

Besides general questions, e.g. which programming languages and IDEs the participants used, it collected information about the usage of the following IDE functionalities:

- Navigation of code
- Debugging
- API and language documentation
- Autocompletion
- Project structure and scaffolding
- Asynchronicity
- Syntax Highlighting

For each of the areas the survey asked if and how the participants were using them and—if appropriate—how their IDE was supporting them. The survey instrumented multiple-choice questions with an additional „Other“ field for custom answers, as well as open-ended questions with a free-form text field.

4.1.1 Survey Results

The survey participants listed 21 different programming languages they are using on a regular basis, as well as experience in 19 different development environments. The participants' background is diverse, although the major part seems to be working with web technologies (both front-end and back-end).

About 76% of the participants look up documentation mainly on the web, only a small percentage uses the integration of documentation into IDEs. However, nearly every participant (87%) makes use of the IDE-provided autocompletion feature, although most of them came up with ways to improve it. Many comments are directed towards „smarter", more context-aware autocomplete suggestions, up to levels of artificial intelligence. Some comments also mention a lack of performance and subtlety.

Navigation within large code bases is done in many different ways, such as presented before: file browsers, symbol browsers, file search or content search. However, there does not seem to be a clear general preference. For structuring code, most participants rely on platform-given modularity, for example through packages and classes in Java. In programming languages where project structures are not given, developers use frameworks and design patterns to achieve a similar structural consistency.

All participants value syntax highlighting, although for different reasons. However, some offered suggestions on how highlighting of certain code tokens could be used otherwise to reduce errors. Two suggestions were targeted towards highlighting of *similar* identifiers in order to recognize typos. Others, however, intended to focus on semantics instead of syntax; for example, indicating value changes of the *this* keyword in JavaScript, highlighting the currently focused block of code, or colour-coding the relationship of interdependent variables.

4.1.2 Interview Results

In succession to the survey, ten participants agreed to be interviewed, seven of which the author conducted interviews with. The interviewees are either currently working as full-time or part-time professional developers or have been doing so in the past and are now in related positions such as *IT consultant*. Aside from that, their backgrounds are diverse, ranging from part-time front-end developers with a focus on design, over web application developers to low-level audio specialists. They have experience with 12 different IDEs, using 15 programming languages on

three different operating systems. Below is a summary of interview results that are related to the thesis project.

Nearly all the interviewees stressed the importance of *performance* in any software development tool, especially in IDEs and code editors. If a feature is too slow, reacts too slowly or slows the overall IDE down, it is considered obtrusive and disturbing to the development workflow. Especially the web developers praised lightweight code editors, favouring them over the more heavyweight IDEs, but still recognizing their drawbacks: lightweight editors are not as *smart* (see below).

Most of the interviewees also referred to their development tools of choice in regards to the *Unix philosophy*, which—according to Ken Thompson—states that programs should „do one thing and do it well." (Raymond 2003) This thought ultimately leads to modularily designed systems, which was stressed in the interviews in different forms. Most obvious is the ability to enable and disable features, as well as some kind of plug-in management in general. Two interviewees also mentioned *modes*, although in different contexts. On the one hand, features could run in different modes to provide help or stay out of they developer's way (*beginner and expert modes*), on the other hand modes could be used to get a different perspective on a program (e.g. highlighting of different aspects in the code).

The interviewees expect their development environment to behave in a *smart* way; it should ideally know beforehand what the developers need in terms of support in a given situation, and what code they are about to write. On first look, and given the current landscape of IDEs and text editors, this contradicts the desire for a lightweight, fast, unopinionated development environment. To be smart, a development environment must have knowledge about the programming language, the libraries used, and about best practices. Some IDEs are tightly integrated with their target programming platforms, for example Microsoft Visual Studio with the .Net platform and Eclipse with the Java platform. But these programs are generally not considered lightweight, performant or unopinionated. *Smartness* for lightweight environments, however, can be achieved by combining it with the modular approach of the Unix philosophy. If specialized programming language tools can be loosely plugged into lightweight development environments, smartness can be achieved in those environments as well. A good example for this is given by the numerous *Linter* plug-ins for editors like Sublime Text (see section **??**: **??**) and projects like CTags[1].

[1] See `http://ctags.sourceforge.net/`

The last relevant result of the interviews to be mentioned in this section is a *focus on code*. No matter the target platform and the developer's background, code is still in focus of the development process nowadays, which makes the code editor the most important part of any development environment. The term *inline* describes activities that happen within the text editor itself, for example syntax highlighting. If development tools work inline, the developer does not have to switch focus back and forth from the authoring process. However, by putting additional information inline, there is a risk that the text editor becomes too cluttered or visually busy, confusing and distracting the developer. This is exemplified by pop-up windows that block a lot of editor space or additional coloured text that makes colour-coding ambiguous. Thus, programming tools that display information inline must be carefully designed to be unobtrusive.

Four important characteristics for programming environments can be extracted from the interviews: *performance*, *modularity*, *smartness* and a *focus on code*. Integrating these characteristics is important for the usability and usefulness of development environments, and thus for any tools that enhance them.

4.1.3 Scope as a valid problem

Through the conducted interviews and the survey, one can argue that *scope* is a promising and valid problem area to explore. Although it was not referred to in the survey in any way, *scope* was mentioned independently by several of the survey participants and interviewees in suggestions for the improvement of existing patterns and tools. One of the interviewees introduced Crockford's (2013) approach of *context colouring* (see below). A similar approach was suggested in the survey in the context of editing. Though not necessarily related to scope, the participant suggested to highlight the current code block the cursor is placed in; this is already done by some editors and IDEs, and is adapted in the concept presented in this thesis as well. Another interviewee suggested to indicate changes of the *this* context in JavaScript, which is closely related to scope, although being run-time specific.

The strongest alternative problem to possibly focus on was *asynchronicity* and the writing and debugging of asynchronous code. After some research on the topic, I found the work of Lieber, Brandt & Miller (2014) to be quite substantial and possibly parallel to my then-prospective work. Lieber implements *Theseus*, an asynchronous JavaScript debugger, and will discuss it from an Interaction Design perspective in his forthcoming master's thesis. This is why I did not choose the topic of asynchronicity, but instead focused on the problem of scope.

4.2 Solutions to analogous problems

The most ubiquitous visualization of program structure is probably *syntax highlighting* or *syntax colouring*. This concept aims to make the developer distinguish entities of the programming language by showing them in different font types, weights, styles, or colours. According to the survey results (see section 4.1: *Survey & Interviews*), syntax highlighting can help with a number of different problems: recognizing errors and typos, distinguishing language constructs from variables and values, and orienting through specific visual patterns. Figure 5.1 shows syntax highlighting in an HTML document; HTML elements are printed in blue, whereas attributes are printed in purple, values in red, comments in yellow, and content in black.

```
1  <!DOCTYPE html PUBLIC "-//W3C/DTD HTML
2  <html>
3      <head>
4          <title>Example</title>
5          <link href="screen.css" rel="sty
6      </head>
7      <body>
8          <h1>
9              <a href="/">Header</a>
10         </h1>
11         <ul id="nav">
12             <li>
13                 <a href="one/">One</a>
14             </li>
15             <li>
16                 <a href="two/">Two</a>
17             </li>
```

Figure 4.1: Syntax highlighting in an HTML document

In his talk „Monads and Gonads“, Crockford presents an alternative to syntax highlighting which he calls „context colouring“ (2013). Instead of using font styles and colours in order to highlight different elements of the *syntax*, he instead highlights different *scopes*. Figure 4.2 illustrates this concept: The global scope is presented in white, whereas nested scopes are marked green, yellow and blue, respectively. In this concrete example, identifiers are always coloured in the colour of the scope of *where they were defined*. For example, the appearance of `value` in the innermost scope is yellow, the colour of the scope in which `value` was declared (as a function parameter to the function `unit`).

```
function MONAD() {
    return function unit(value) {
        var monad = Object.create(null);
        monad.bind = function (func) {
            return func(value);
        };
        return monad;
    };
}
```

Figure 4.2: Context colouring in JavaScript, as proposed by Crockford[1]

Theseus is a JavaScript debugger built as a plug-in for the Brackets IDE. It makes use of the code editor itself and shows information inline, in the gutter and in a panel on the bottom of the Brackets window (see Figure 4.3). Theseus is mostly used for asynchronous debugging, so the way those UI elements are used corresponds to this purpose. For every function definition, Theseus shows the number of times the function has been called in the gutter. Functions that have never been called are marked with a grey background in the source code. Additionally, the panel on the bottom shows information about the function the cursor is positioned in[2].

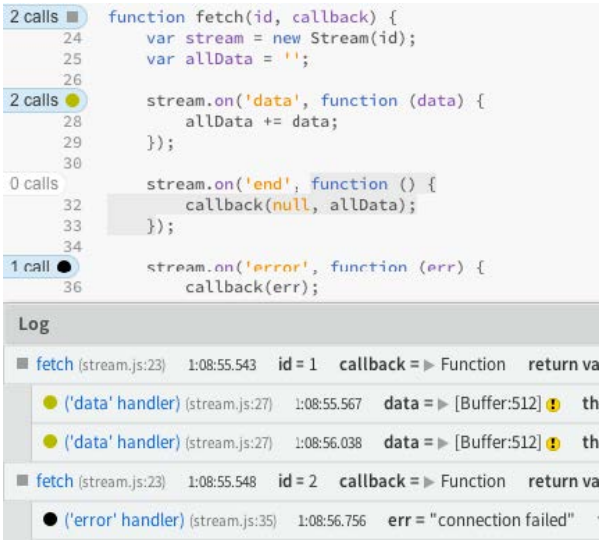

Figure 4.3: Theseus' asynchronous JavaScript debugging (Lieber et al. 2014)

JSHint is a so-called *linting* tool, or *linter*, for JavaScript: it detects bad programming practices (*code smells*) by checking JavaScript code against a set of rules, and therefore tries to prevent com-

[2] It shows asynchronous call stacks, which are not of relevance to this thesis.

mon problems. Originally built as a command-line tool and for online code checking, JSHint is implemented in many IDEs through the respective plug-in systems. The *Sublime Linter* plug-in[1] for Sublime Text 3 implements JSHint (and other linting tools) *inline*: hints of bad code or inconsistent style are shown in the text editor itself and are indicated in the gutter. If the cursor is on top of problematic code, the respective hint is printed in the status bar. *Sublime Linter* behaves according to the characteristics identified before: it is modular, as linters for different programming languages can be plugged-in; it is lightweight and does not slow the editor down; it focuses on code by displaying results inline without cluttering the editor window; and it is smart to some extent, as it allows the configuration of certain programming guidelines. The prototype presented later in this thesis borrows many characteristics and design decisions from linting tools such as this one. This is due to the fact that both the problem of code smells and the problem of scope can mostly be solved with static analysis[2] and presented in a similar manner.

(Sublime Linter screenshot)

In terms of navigating and displaying tree structures in relation to the actual source code, the *Element Inspector* of **Chrome Developer Tools** makes a good example (see Figure 4.4). The structure of HTML is quite similar to that of scope, as it is nested in the same way, which is why ideas can be taken from the Developer Tools. They show the source code of the inspected website and allows the user to select any HTML element within. In the remaining parts of the window, information relevant to the selected element is shown.

At the bottom of the window, a status bar shows the nesting of the selected element: on the visual left (and logical top of the tree) is the `html` element, inside it the `body`, then a `div` and finally the selected `a` element. This status bar can be used to navigate around the nested elements by clicking on them. Clicking on the `body` element highlights it in the source code as well, and shows different style information on the right-hand side.

Placed to the right of the source code is a sidebar. Although it contains a tabbed interface to browse different facets of the selected element, the one that is relevant is the one in focus on the screenshot, *Style*. The way style (through Cascading Stylesheets (CSS)) is applied to HTML elements is similar to the way nested scope works: style that is defined on the containing elements may influence the style of the selected elements, which is why the relevant styles are listed in order of precedence. The style rules that apply with the highest precedence are listed on top, while the rules with the least precedence are listed on the bottom. Style rules that are overriden by rules of higher precedence are striked through to indicate that they do not apply

[1] See `http://www.sublimelinter.com/`

[2] Static analysis is the analysis of computer software that is performed without actually executing the program.

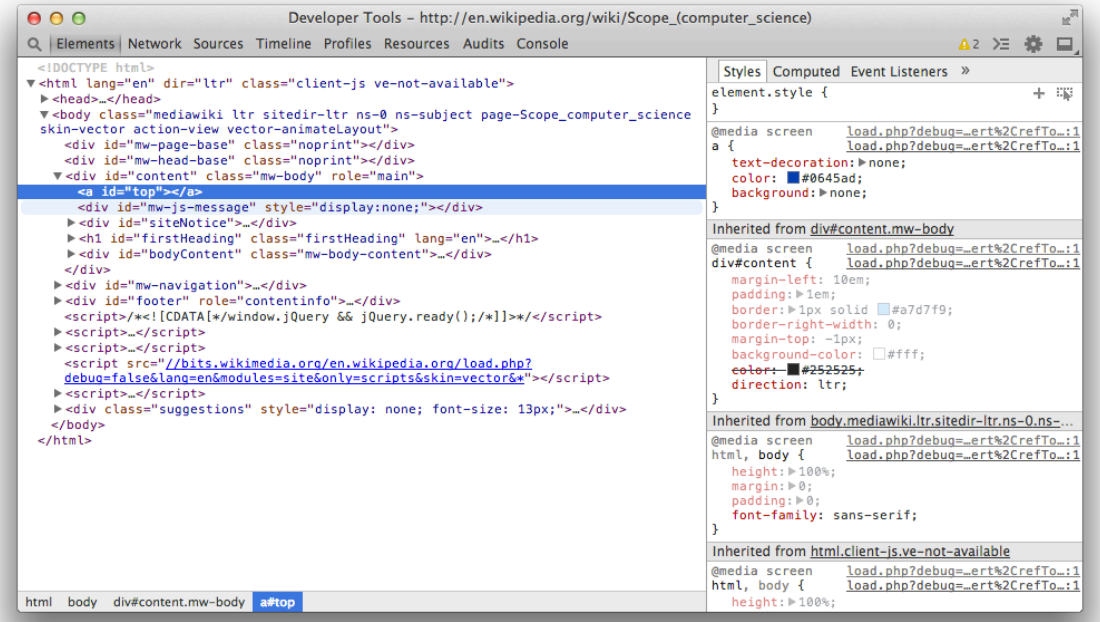

Figure 4.4: Chrome DevTools with Element Inspector

anymore. This way of visualizing and organizing information about nested structures is further used in the following concept and design phases (see section 4.3 and chapter 5).

4.3 Ideation

To support the ideation phase, existing UI components used within IDEs, as presented in chapter 2, were collected. Those components were written down on post-it notes and used as seeds for *seeded brainstorming*: for each of the components, a set of solutions should be developed that are similar, related to or based on the respective component. Most of the ideas that came out of the brainstorming session made use of multiple components, for example the *scope chain* which is described further down: it made use of a status bar as well as the code editor.

Some ideas of the brainstorming phase made it into first sketches. Depending on if the idea depended on actual source code, the sketching took two different approaches. For ideas that involved code, it was important to work with real, functioning code. Therefore, two JavaScript applications were created to serve as examples:

- A small web server application, that parses a markdown-formatted text file and renders it into an HTML template. The application runs on Node.js and represents a typical control flow for e.g. a blogging engine (i.e. content + template = site).
- A client-side script (runs in a web browser) that fetches JSON data and presents them on a website, by the click of a button. This script represents typical client-side UI code, connecting a button event to a function and presenting the result in the UI.

The two applications were written in different styles: the server-side application decouples its tasks by putting them into specific, named functions (as far as it was seen appropriate) and therefore has a relatively flat tree structure. The client-side application, however, nests all function definitions inside each other, resulting in nearly one function definition on each line, and far deeper indentation (in other words, higher code complexity and deeply nested scope). A good solution for this design problem should address both of the cases.

Printouts of the two JavaScript files served as a basis for ideation that involved source code. However, for concept ideas that would mainly work with other UI components, such as a sidebar, or such concept ideas that would introduce new UI components, a blank notebook was used for sketching.

TODO: source code examples will be in the appendix and be referenced here

- *Scope Chain* - a breadcrumb view of the current scope chain, similar to that of a selector chain in an HTML editor (screenshot!). The scope at the position of the cursor would be shown in a status bar. By hovering over a scope level, the corresponding source code would be highlighted in the source editor.
- *Scope Graph* - similar to a class browser, the scope graph would represent a tree view of the application's scopes. This could be implemented as a sidebar or panel.
- *Scope Colouring* - similarly suggested by Crockford (2013), the source code can be coloured in depending on its scope level. Crockfords variation is meant to replace syntax highlighting; one could, as well, complement syntax highlighting by colouring in the background (as e.g. Theseus does), e.g. in shades of grey.
- *Inspect Scope* - comparable to DevTools' *Inspect Element* function, the user can right-click into the source code and choose *Inspect Scope*, which opens a panel that shows global variables, current local variables as well as the value of `this`.
- *Gutter Scope* - any change of scope is indicated in the code editor's gutter (similar to JSHint).
- *Quick Inspect* - similar to Brackets' *Quick Edit* feature, the value of `this` could be inspected inline. Debugger stuff, as *this* is run-time.

TODO: more detail, and scans of relevant sketches (not! all of them)

TODO: Chosen solution is combination of scope chain, scope colouring and inspect scope

5 Design

The following chapter will expose the design process and reasoning behind certain design decisions made for the prototypes. It will lead through the different prototyping stages and user testing to an evaluation of the design, including an outlook.

The prototyping was happening in three subsequent stages. The first stage comprised a set of pencil sketches on paper; the second prototype was created with web technology, but fixed around a certain source code and faked interactivity; the third prototype was implemented as a plug-in for the Atom editor, and is able to work with any source code it is provided.

5.1 Definitions

To be able to talk about the qualitites of the concept and prototypes, we must first define a number of terms.

Scope block In a JavaScript text file, a scope block is the textual block representing a logical scope. For a function, which in JavaScript creates a new scope, the scope block starts at the `function` keyword and ends at the closing curly brace `}` of the function body. If, in a text editor, the cursor is placed anywhere inside this scope block (but outside of child scopes), the scope block is called *active scope*.

Current scope In a running JavaScript program, it is the currently executed scope. This is a term related to the run-time rather then to author-time, and should not be confused with *active scope* described below.

Active scope The scope which is currently in focus of editing. In relation to IDEs, code editors and the prototypes presented in this chapter, the active scope always describes the scope that the cursor is placed in.

Local scope In the context of nested scopes, the local scope is the one in focus (be it in the execution context during run-time, or the editing context during author-time). Local scope is contrasted with non-local scope; scopes that are logically distant from the local scope. Those may be ancestor scopes, descendant scopes, or parallel scopes. The term is also used to contrast *global scope*.

5.2 Sketching

One could argue that sketching is part of the earlier exploration phase, rather than of the prototyping phase. However, next to sketching different ideas, the author also sketched different possible implementation for one feature that seemed valuable to the design solution: *highlighting*.

The basis for the sketches were printouts of the same source code, each leading to a different way of highlighting.

(scans here)

Active scope, inclusive

This sketch highlights the active scope block by applying a background colour to it. The highlighting is *inclusive*, i.e. any descendant (inner) scopes are highlightes as well.

Active scope, exclusive

Same as above, but descendant scope blocks are excluded from highlighting. This way of highlighting was implemented in the scripted prototype (see **??**).

Active scope and ancestor scopes

Next to highlighting the active scope, its ancestor scopes can also be highlighted to emphasize nesting. To contrast the ancestor scopes from the active scope, the highlighting would make use of different background colours, for example different shades of grey. This way of highlighting can be combined with both the inclusive and exclusive approach.

Scope colouring

Described by Crockford (2013) as „context colouring“, this way of highlighting would not apply a background colour, but instead replace the existing forms of syntax highlighting. Thus, the highlighting would not depend on the cursor position (which defines the *active scope*), but would be static instead.

Identifier origin

Additionally to emphasizing code blocks, individual identifiers can be highlighted. Given a highlighted active scope, this sketch highlights identifiers that are defined in that scope but used somewhere else (in descendant scopes).

This works as well for the *scope colouring* described above, as each scope has a fixed colour. Identifiers that are used in other scopes than they are defined in can therefore always be recognized if they appear in the colour of their origin scope.

5.3 Scripted prototype

It very quickly became clear that the sketches were of little value. Although most of them gave a general impression on where the selected scope started and where it ended, it did not allow the user to see the big picture. It seemed probably that a more interactive prototype would be more helpful in this regard. As its capabilities of working with code are limited and the code had to be specifically prepared, this is called a *scripted prototype*.

As the author is most familiar with web technologies, the scripted prototypes would be built using Hypertext Markup Language (HTML), CSS and JavaScript and run in a web browser. Other prototyping tools, such as Balsamiq or Indigo Studio, would not allow for enough detail in terms of highlighting certain code passages, and would have represented a learning overhead.

A code syntax highlighter[1] was used to turn the subject source code into styled HTML tags, to make it appear as if it was inside of a real code editor. Applying syntax highlighting was also necessary to see if the different highlighting techniques, as sketched out in the previous phase, would interfere with syntax highlighting.

[1] Prism, see `http://prismjs.com/`

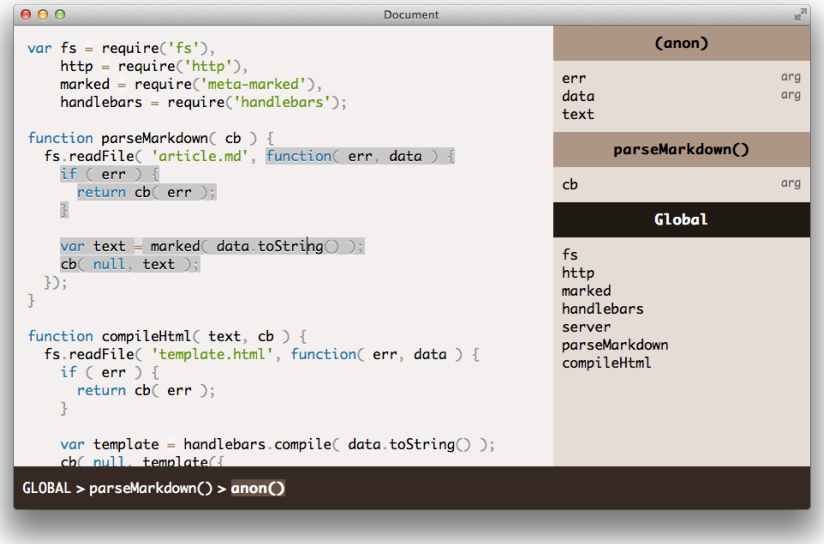

Figure 5.1: Screenshot of the scripted prototype run in a browser

Furthermore, markers in the form of HTML tags were added to the subject source code, which made it possible to apply different stylesFor example text colours, background colours, of font styles. to regions of the code. This was later used to realize highlighting of the .

Two distinct UI elements were added: a sidebar and a bottom bar. The content of both depends on the *active scope*, i.e. the scope the cursor is placed in.

For each of the nested that the cursor is positioned in (beginning from the local scope, going outwards up to the global scope), the sidebar shows a pane. Each pane contains the scope's name along with a list of identifiers defined within that scope. In opposition to the working prototype, the scripted prototype does not show phenomena like hoisting and shadowing. The panes are ordered ascending by logical distance, i.e. the local scope would be on top, the next surrounding scope beneath it, and so forth; up to the global scope on the bottom.

The different panes are hardcoded: all panes exist in the markup of the prototype at all times and are pre-filled with the relevant data, but are shown and hidden on demand.

The bottom bar shows a horizontal list of scope names. It makes use of the *breadcrumbs* UI pattern[1]. Each listed scope, beginning from the global scope on the very left, up to the local scope on the right, can be highlighted and navigated to by clicking on its label. By hovering[2]

[1] In the Yahoo pattern library: `https://developer.yahoo.com/ypatterns/navigation/breadcrumbs.html`

[2] Hovering: Placing the mouse cursor over an element.

over the label, the user can get a preview of the target scope, as it is highlighted in the editor alongside the currently active scope.

5.3.1 Constraints

The scripted prototype has some drawbacks, some of which might influence its quality. The most obvious one is the fact that it only works with a static, predefined source code, which is manually adapted to serve the prototype's purpose. This implies that

1. changes can not be made to the code, which makes the experience of the prototype very different from a real code editor, and
2. it is hard to tell if the prototype works similarily well with code that is more complex, less complex, or of an overall different style.

The fact that there are no text editing facilities comes with another drawback, namely the absence of a cursor. If a cursor cannot be placed anywhere in the editor, the „activation" of a scope block must be achieved differently. In the case of this prototype, it is solved by clicking on a piece of code. However, clicking anywhere in the line besides the actual text will not change the active scope.

5.3.2 Evaluation of the scripted prototype

The prototype was tested with two JavaScript developers in individual in-person walkthrough sessions. The users were introduced to the concept, if they were not familiar with it already, and explained the basic constraints of the prototype (as mentioned above). They were thereafter able to explore and test the prototype to their likings. One of the two sessions have been recorded as a screencast.

The users liked both the „preview" feature (hovering over a breadcrumb) and the sidebar. The preview gave them an opportunity to quickly get a visual overview of your position in the code and the active scope chain. The sidebar showed them which variables and functions are available in a given context. Overall, they liked the dynamicity of the prototype, as they could „play around" with it and understand the design concept just by trying. They quickly made a connection between they position of the cursor and the active scope along with its content.

One of the users suggested possible improvements or alternative designs for existing features. He recommended a wider use of colour coding to create a link between the scope in the editor window and the sidebar, for example by colouring all the ancestor scopes in different shades of grey. He also suggested an alternative visual structure for the sidebar instead of the list, for example nested clusters or a graph. For hoisting, the user came up with an idea to integrate indicators into the text editor: a „phantom“ variable declaration, which would be grey and not editable, could be inserted on top of a scope, to show that a certain variable declaration would be hoisted up there. This indicator should be collapsible so that it does not interfere with the editing process. Because of technical constraints, this idea was not implemented in the next prototype iteration; however, it seems a sensible solution to the hoisting indication problem, as it communicates this implicit phenomenon very clearly.

In conclusion, the prototype was well-received and served its purpose well. It became clear that a consistent and clear visual language for the next iteration of the prototype was necessary, and that a direct connection between the code and the scope visualization has to be communicated.

5.4 Working prototype

The second prototype, which emerged into the final one, was built as a working prototype capable of handling any JavaScript scope, rather than as a proof-of-concept. It was integrated into the Atom[1] text editor as a so-called *package*, released as Open Source Software (OSS) and was made publicly available for using and testing. The package is called „Scope Inspector“ and will be referred to using this name throughout this section.

5.4.1 The prototyping platform

For the prototype to yield meaningful results, I decided to integrate it into a real IDE. This decision was informed by several circumstances. The first and most obvious is that I could address a broader community of users this way. If the prototype was, like the scripted prototype, implemented in isolation as a standalone application, it would raise the barrier for people to test and for me to distribute it. But by building it as a plug-in to an existing IDE, I could leverage the distribution channels that were already in place. A second reason is that users are already

[1] See `https://atom.io/`

familiar with the software and do not have to orientate themselves anew. This also implies that all the features that the user *expects* from an IDE are in place, and the prototype can more seamlessly be integrated into the user's workflow. Finally, the third reason for building a prototype on top of an existing IDE is that it can make use of the design language in place, which eliminates the need to take decisions that are rather irrelevant for this prototype, such as the choice of a typeface and colour palette.

As a prototyping platform, the author decided on the Atom text editor. Atom is open source and created by the software company Github. By the time of writing, Atom is a relatively young project with a growing community and plug-in ecosystem. The reasons for deciding in favour of Atom are threefold: the technologies it is built upon, its internal software architecture, and the user group it is targeting.

Atom is built on web technologies, namely WebKit[1] and Node.js[2]. WebKit is the browser engine used by the web browsers Google Chrome and Apple Safari, amongst others, and is therefore responsible for the User Interface layer of Atom. Node.js is the JavaScript platform responsible for running any non-UI logic. Atom is mostly written in CoffeeScript[3]. Consequently, Atom packages can be written in CoffeeScript or JavaScript, using HTML and CSS for the UI. As I am familiar with these technologies, Atom provided an ideal prototyping platform with a low entry barrier.

For extending Atom, it offers an Application Programming Interface (API) which can be used by plug-ins. Atom's internal architecture is built in a modular way, so that plug-ins can hook into nearly everything that happens and react on it. The prototype makes use of this fact in many ways, for example by showing and hiding its UI elements depending on the type of file that is being edited. In general

Atom is marketed by Github as a „hackable text editor for the 21st Century"[4]. It is also intended to be a „deeply extensible system that blurs the distinction between ‚user' and ‚developer'." Those claims lead to the conclusion that Atom is a text editor built for developers, especially—but not exclusively—web developers. While not every web developer is a proficient JavaScript developer, the target groups of Atom and this prototype seem to overlap to a large extent.

1 See `http://www.webkit.org/`
2 See `http://nodejs.org/`
3 CoffeeScript is a programming language that transcompiles to JavaScript.
4 See `https://atom.io/`, accessed 18.05.2014

5.4.2 Parsing and gathering relevant information

For the prototype to be as *complete* and *correct* as possible, it was built on top of an existing JavaScript parser called Esprima[1]. The process of extracting the relevant scope structure and annotations from the Abstract Syntax Tree (AST)[2] will not be discussed here in greater detail, but is instead described in a blog post (von Oldenburg 2014).

However, it is important to mention what data structures are extracted from the source code. Analoguous to the nature of JavaScript scope as described in chapter 2: *Theoretical Grounding*, the data structure is a hierarchy of objects. Each object represents a scope and may have metadata as well as a list of identifiers attached to it. An identifier is either a child scope (as created by a function) or a variable. For scope objects, the metadata are its name and its location in the source code (row and column of the start and end points), whereas for variables the metadata are its name, location, if it is hoisted, by which child scope identifiers it is shadowed, and which identifier it is shadowing.

A diagram of an exemplary data structure is shown in figure **??**.

TODO: insert diagram of data structure here

Using this data structure, the prototype can show meaningful data to the user, which would not have been possible with the AST alone. The modular composition of the prototype, which decouples the task of parsing from the task of displaying information, makes it possible to re-use each of the components. The component described in this section, which is responsible for turning the AST into a „scope tree“, could be used in plug-ins for any IDE to achieve similar functionality as the one of this prototype.

5.4.3 Interface and Interactions

The design respects that the subject of a developer's work is the code itself, not the tools that surround it. This is why the solution integrates into the most important part of the IDE, the text editor, directly. The features built into the editor itself will be called *inline* features.

Atom's interface is, by default, threefold: the text editor takes the most space; on its left is a sidebar containing a file browser, and on the bottom is a status bar. As many web browsers, text

[1] See http://esprima.org/

[2] The AST is the data structure that is returned by the parser, which contains all the lexical statements and expressions.

editors, and IDEs, multiple open files are accessed through *tabs* on the top of the screen. The tabs are important, because the Scope Inspector will only be active as long as an editor with a JavaScript file is in the foreground. Whenever the user switches to another tab, the Scope Inspector is activated or deactivated, depending on if the tab contains an editor with a JavaScript file or not.

Throughout the Scope Inspector package, a visual style consistent with Atom's is used. Any icons in use are taken from the Octicons[1] icon set, which is incorporated into every Github product. Atom supports themes (colour schemes) for both the application window and the editor. Scope Inspector makes use of the colours defined in those themes. This way, the package UI feels more natural to the user. However, there may be difficulties if the theme is not well-defined and the colours are badly balanced. One user reported very low contrast between the editor's background and the scope highlighting. In addition to pre-defined theme colours, Atom also provides a set of pre-styled UI components, for example buttons and panes, which have been used in the prototype.

Whenever the Scope Inspector is active, two things are obvious: on the bottom of the editor, a panel is shown which we call *bottom bar*, and the active scope is highlighted inline. Additionally, a sidebar can be toggled using the Atom command „Scope Inspector: Toggle Sidebar". This command is accessible using the menu, the command palette, a keyboard shortcut (`Ctrl+Alt+i` by default), and a toggle button on the bottom bar. The several components and their functionality are explained in more detail in the following sections.

Inline scope highlighting

As explained above, the *active scope* is the immediate scope the cursor is placed in. It is emphasized by highlighting it through a lighter or darker background colour (depending on Atom's colour scheme). If the cursor is placed in a different scope, the formerly active scope is unhighlighted, and the now active scope is highlighted instead.

While the scripted prototype implements *exclusive highlighting*, this prototype now implements *inclusive highlighting*, which means that the inner scope are highlighted as well. This is due to technical reasons; building exclusive highlighting into the prototype would have taked a lot more time. In further iterations of the prototype, an option to enable and disable exclusive highlighting could be provided.

[1] See `https://github.com/styleguide/css/7.0`

The bottom bar contains a toggle button[1] to enable or disable highlighting of the global scope. Highlighting the global scope with *inclusive highlighting* is not useful, as the whole file would be highlighted (and there would be nothing left to contrast the highlight to).

Bottom bar

The bottom bar serves two purposes: it provides a quick glance of where in the scope hierarchy the cursor is and provides quick access to two settings.

On the right side of the bottom bar, to toggle buttons allow for enabling and disabling of two features. The right button, showing a list icon, shows or hides the sidebar. The left button with the label „Highlight Global“ toggles the highlighting of the global scope (as described above).

The left side of the bottom bar shows the breadcrumbs known from the scripted prototype. The breadcrumbs, implemented as simple buttons, are labeled with the corresponding scope name. The global scope is always on the left, whereas the currently local, active scope is on the right. By hovering over any of the breadcrumb buttons, the user can preview the respective scope highlighting in the editor. The preview is applied in addition to the currently active highlight in a different colour.

By hovering over the breadcrumbs from left to right or from right to left, the user can make the relationship between the logical structure of the JavaScript program (in the form of hierarchic scopes) and the textual structure (in the form of code) visible.

Sidebar

The sidebar shows content depending on the currently active scope. Similarily to the scripted prototype, the sidebar lists one pane for each scope in the hierarchy of the active scope. The active scope is listed on top, while its ancestors are listed below, up to the global scope on the very bottom.

Each pane is entitled by the name of the scope. In case of function scope, the name of the function becomes the scope name („(anonymous function)“ in the case of an unnamed function expression). In case of the global scope, the name is „GLOBAL“.

Underneath the title, the names of all identifiers defined within the scope are listed, along with certain attribute annotations.

[1] A switch in the form of a button, which can be either *on* or *off*.

- Function parameters are listed first. They appear with the annotation „param“, set in smaller text size to the right.
- General variables follow the parameters. If they are not shadowed, they have no annotations.
- Functions are the last entities in the list. They are connotated with a pair of parantheses „()“.

The listed identifiers show also if they are hoisted, shadowed, or if they shadow other identifiers. This is indicated by different stylistic changes.

- Hoisted identifiers have a small, upwards-pointed arrow on the left side of their label. This indicates that their declaration is implicility moved upwards in code.
- Shadowed identifiers are printed in a more subtle text colour. Besides that, their label is striked-through to indicate that they are not accessible within the given descendant scope.
- Identifiers that shadow other identifiers in ancestor scopes are printed in a highlight colour. In case of Atom's standard UI theme, this is a bright blue colour.

5.5 User Testing & Evaluation

The goal of user testing was to collect both qualitative and quantitative data through different methods. The quantitative data collection was built into the prototype in form of a connection with Google Analytics[1].

5.5.1 Test installment

Atom includes a package management system with an online repository, called Atom Package Manager (APM). This system allows any developer to publish Atom packages and thus make them available for any Atom user to download and use. Consequently, this prototype was distributed via APM.

The author was collaborating with two full-time developers and one part-time developer. They agreed to install the package and use it over the course of one week (full-time developers) or one day (part-time developer), respectively, integrating it into their usual workflows.

[1] A website and app analytics platform.

In addition to this directed user test, publishing the prototype via APM made it available to the general public. It was announced on several social networks, especially targeting existing Atom users, with the goal of getting users to download and use it. A week after publishing the prototype, the number of downloads counted ???. This way of testing „in the wild“ makes it harder to gather feedback, compared to the method of addressing potential users directly. However, the analytics mechanism built into the prototype yielded quantitative data for evaluation.

5.5.2 Usage metrics

The prototype was built with the option to collect usage metrics using the Google Analytics service. By default, this option was set to *off*, as the author opposes the unknown tracking of any data. However, users were asked to enable tracking in Atom's *settings* panel for the Scope Inspector package.

If enabled, the following events are tracked:

- The package is enabled/disabled
- The sidebar is shown/hidden
- The user hovers over a scope breadcrumb in the bottom bar and thus previews a scope highlighting
- The user clicks on a scope breadcrumb in the bottom bar and thus jumps to the beginning of scope, making it the active scope and highlighting it

Through collecting these events, a claim can be made—to a certain extent—for how helpful certain features are.

5.5.3 Evaluation / Interviews

Analytics results / metrics

Interviews with developers

6 Conclusion, Reflection

Bibliography

Baxter-Reynolds, M. (2011), 'Program in a text editor rather than an ide? why would you do that?'.
URL: *http://www.theguardian.com/technology/blog/2011/oct/24/programming-ide-editors-choice*

Castorina, D. (2014), 'Javascript prototypes, scopes, and performance: What you need to know'. Accessed: 16.05.2014.
URL: *http://www.toptal.com/javascript/javascript-prototypes-scopes-and-performance-what-you-need-to-know*

Crockford, D. (2013), 'Monads and gonads'. Accessed: 26.04.2014.
URL: *https://www.youtube.com/watch?v=b0EF0VTs9Dc*

Interaktives Programmieren als System-Schlager (1975). German. Accessed: 28.04.2014.
URL: *http://www.computerwoche.de/heftarchiv/1975/47/1205421/*

Lieber, T., Brandt, J. & Miller, R. C. (2014), Addressing misconceptions about code with always-on programming visualizations, *in* 'Proceedings of the SIGCHI Conference on Human Factors in Computing Systems', CHI '14, ACM, New York, NY, USA, pp. 2481–2490.
URL: *http://doi.acm.org/10.1145/2556288.2557409*

Lynch, D. (2011), 'Why not just use an ide if you want ide features?'.
URL: *http://davidlynch.org/blog/2011/09/why-not-just-use-an-ide-if-you-want-ide-features/*

McConnell, Steve (2004), *Code Complete: A Practical Handbook of Software Construction*, 2nd edn, Microsoft Press.

Moggridge, B. (2007), *Designing Interactions*, The MIT Press, Cambridge, MA, USA.

Raymond, E. S. (2003), *The Art of Unix Programming*, Addison-Wesley.

Simpson, K. (2014), *You Don't Know JS: Scope & Closures*, O'Reilly Media.

Victor, B. (2012), 'Learnable programming'. Accessed: 07.04.2014.
URL: *http://worrydream.com/#!/LearnableProgramming*

von Oldenburg, T. (2014), 'Analyzing javascript scope'. Accessed: 12.05.2014.
URL: *http://www.protoandtype.com/analyzing-scope/*